



Technical Report
CMU/SEI-96-TR-035
ESC-TR-96-035
Carnegie-Mellon University
Software Engineering Institute

A Case Study in Structural Modeling

Gary Chastek

Lisa Brownword

December 1996

19970128 058

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DATA QUALITY UNPUBLISHED

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of, "Don't ask, don't tell, don't pursue," excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

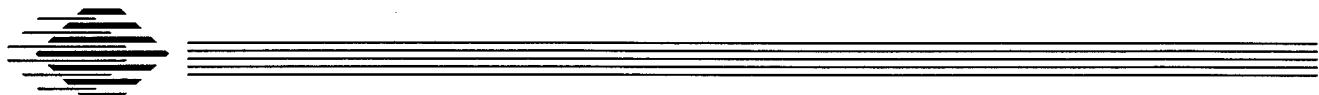
Technical Report

CMU/SEI-96-TR-035

ESC-TR-96-035

December 1996

A Case Study in Structural Modeling



Gary Chastek

Lisa Brownsword

Product Line Systems Program

Unlimited distribution subject to the copyright.

Software Engineering Institute

Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Copyright © 1996 by Carnegie Mellon University.

Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and "No Warranty" statements are included with all reproductions and derivative works.

Requests for permission to reproduce this document or to prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is <http://www.rai.com>

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1	Introduction	1
1.1	Purpose of the Case Study	1
1.2	Motivation	2
1.3	Information Gathering	2
1.4	Report Organization	3
2	Architecture-Based Development	5
2.1	Introduction	5
2.2	Software Architecture	5
2.3	Architecture-Based Development	6
3	Aircrew Trainer Simulation Software	9
3.1	Introduction	9
3.2	Aircrew Trainers	9
3.2.1	Overview	9
3.2.2	Associated Benefits	10
3.3	Aircrew Trainer Simulation Software	11
3.4	Data-Driven Software Architecture	13
3.4.1	Overview	13
3.4.2	Difficulties with the Data-Driven Architecture	14
3.5	The B-2 Project	15
3.6	Summary	16
4	Structural Modeling	17
4.1	Introduction	17
4.2	Separation of Domain Commonality from Variability	18
4.3	Object-Based Components	18
4.4	Structural Types	20
4.5	Subsystems	20
4.5.1	The Subsystem from the Outside	21
4.5.2	The Subsystem from the Inside	22
4.6	Structural Models	23
4.7	Development Artifacts	24
4.8	Summary	25
5	Organizational Factors	27
5.1	Introduction	27
5.2	Results	28
5.3	Impact on Organizational Structure	29
5.4	Impact on Staffing Profiles	31
5.5	Impact on Development Approach	32

5.5.1	Life Cycle	32
5.5.2	Methods and Tools	33
5.5.3	Documentation	34
5.6	Managing the Learning Curve	35
5.7	Summary	36
6	Analysis and Conclusions	39
6.1	Introduction	39
6.2	Limitations of the Case Study	39
6.3	Organizational Factors	39
6.4	Architecture-Based Development	40
6.5	Structural Modeling and Object-Oriented Development	41
6.5.1	Overview of Object-Oriented Development	41
6.5.2	Similarities	42
6.5.3	Differences	42
6.6	Product Line of Air Vehicle Simulation Software	43
Appendix A	The Executive	45
Appendix B	Control Flow	49
References		53

List of Figures

Figure 1:	Operation of an Aircrew Trainer	10
Figure 2:	Simulator Time Period	12
Figure 3:	Data-Driven Software Architecture	13
Figure 4:	High-Level Partitioning of Simulator Software	18
Figure 5:	Partitioning of the Aircraft-Specific State Calculations	19
Figure 6:	Component Structural Type	20
Figure 7:	View Outside of the Subsystem	21
Figure 8:	Subsystem Controller Structural Type	22
Figure 9:	Reported Defects - Before and Since Using Structural Model	29
Figure 10:	Executive Partitioning	45
Figure 11:	Periodic Sequencer	46
Figure 12:	Event Handler	46
Figure 13:	Surrogate	47
Figure 14:	System Time Period	47
Figure 15:	Flow of Control: Periodic	50
Figure 16:	Flow of Control: Aperiodic	51

A Case Study in Structural Modeling

Abstract: This report is one in a series of Software Engineering Institute (SEI) case studies on software architecture. It describes structural modeling, a technique for creating software architectures based on a small set of design elements called structural types. Structural modeling resulted from the efforts of the Air Force Aeronautical Systems Command (ASC/YW) and has been used by Air Force contractors since the late 1980s to design large-scale, high-fidelity aircrew trainer simulation software. This report examines the changes, resulting from the use of structural modeling, to the trainer's software architecture and to the development methods used.

1 Introduction

1.1 Purpose of the Case Study

Each SEI case study in software architecture documents a specific organization's process for selecting, evaluating, and using a software architecture.¹ The goal of these studies is to allow the reader to compare the different organizations, and the software architectures adopted. Issues addressed include

- How is the software architecture used to develop a software system?
- What are the benefits of the approach taken?
- What are the associated costs and difficulties?

This report focuses on a joint effort, led by the U.S. Air Force Aeronautical Systems Command (ASC/YW) and supported by Air Force contractors and the Software Engineering Institute (SEI), that resulted in structural modeling. ASC/YW is the Air Force organization responsible for the acquisition of aircrew trainers, including the selection of the contractors to design and build a trainer, coordination of the trainer's construction, and selection of the trainer's maintenance contractor.

The initial goal of the joint effort was to reduce the complexity of producing aircrew trainer simulation software; particularly complexity as encountered during software integration. Because structural modeling has been used across multiple aircrew trainers, the lessons learned from its use have strong implications for the creation of product lines of aircrew trainers.

This paper describes structural modeling, the aircrew trainer simulation software that was designed using structural modeling, the software architecture of the trainers, the technologies

¹. The other currently published case study in this series is *Case Study in Successful Product Line Development* [Brownword 96].

used to develop the trainers, and the organizational, management, and business issues related to structural modeling.

The intended audience for this case study includes those interested in software development, particularly change agents, technology champions and sponsors, as well as program and acquisition managers from government and industry. We assume that the reader has no previous in-depth knowledge of simulation software, trainers, software architecture, or architecture-based development; these terms will be defined and discussed briefly.

1.2 Motivation

The U. S. Air Force has a long history of using aircrew trainers as an integral part of their aircrew training programs. These trainers are expensive to build. For example, the B-2 trainer contains over 1.7 million lines of Ada simulation code. However, the benefits of trainers justify the expense. Aircrew trainers reduce training costs, improve safety, support security, and provide flexibility and convenience [Rolfe 86].

Modern aircraft have become increasingly complex and dynamic; an aircraft is dynamic if its systems (e.g., weapons and radar) are subject to frequent change. Aircrew trainer simulation software based on the traditional data-driven approach have become increasingly difficult to integrate and maintain. (A description of the data-driven approach is contained in Chapter 3.) Structural modeling ultimately addressed both the technical and developmental problems associated with the earlier efforts to develop trainer simulation software.²

Over the past decade the U. S. Air Force has embraced the use of structural modeling, funded its use on several programs, and ultimately sponsored the creation of a guidebook on structural modeling [ASCYW 94].

1.3 Information Gathering

The information presented in this report was gathered from on-site visits and phone conversations with aircrew trainer contractors, interviews with SEI employees involved with the structural modeling effort, and publications related to structural modeling.

2. Structural modeling has been in use for over a decade, during which time the technology has matured. The notion of structural modeling has expanded from a method for constructing a software architecture into an architecture-based development method. This report presents the expanded view.

The contractor organizations that we contacted include

- Hughes Training, Inc., Link Division (B-2 Weapons System Trainer)
- Loral (C130 Special Operations Forces Trainer)
- Lockheed Martin (F-22 Trainer)

1.4 Report Organization

The structure of this report is as follows:

- In Chapter 2, we discuss software systems as a solution to a user-defined problem, and define and discuss software architecture and architecture-based development.
- In Chapter 3, we discuss aircrew trainer simulation software. We describe aircrew trainers in general and the required simulation software in particular, and describe the data-driven software architecture and its limitations.
- In Chapter 4, we explain structural modeling as a response to the limitations of the data-driven architecture.
- In Chapter 5, we present the organizational factors related to the use of structural modeling in aircrew trainers.
- Finally, in Chapter 6 we present our analysis of structural modeling based on the gathered information.

2 Architecture-Based Development

2.1 Introduction

A software system is a solution to a user problem. Software design methods typically describe the user problem in terms of the restrictions on a potential solution (i.e., by requirements, goals, and constraints). Generally, the requirements describe the needed features (i.e., functionality) of the software system; the goals specify the required quality attributes (e.g., modifiability) of the software system; and the constraints enumerate the limitations (e.g., cost and schedule) placed on the software system [Strikeleather 96].³

"A successful software system is conceived, created, and maintained by the coordinated efforts of the stakeholders of that system: the customers, users, project managers, architects, coders, testers, etc." [Clements 96]. Each of these stakeholders has a unique perspective on the software system. Together, the requirements, goals, and constraints unify the stakeholders' views into a precise statement of the original user problem.

Just as the requirements, goals, and constraints form a representation of a particular user problem, a software architecture is a representation of a particular solution to that problem. The software architecture can provide the common focus for the successful development and maintenance of a large, long-lived software system. Software development based on the architecture is called architecture-based development. In this chapter we will define and discuss the notions of software architecture and architecture-based development.

2.2 Software Architecture

No single definition of a software architecture is universally accepted: many definitions exist. [Clements 96]. We have adopted the following definition of a software architecture from Garlan [Garlan 95]:

... the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.

In this definition, *the components of a program/system* refer to how the functionality of the software system is assigned to the individual software components; issues include modularity and the granularity of the software components. The *interrelationships* of the components refer to the coordination of the software components (i.e., the coordination of, communication between, and synchronization of the components). The *principles and guidelines* refer to the design decisions made, their rationale, and the policies for adopting and enforcing those

³ In practice, partitioning these restrictions into requirements, goals, and constraints is frequently not as clear cut as we imply. For example, certain system quality attributes, such as performance and security, can be viewed as system requirements. Our intention is to be inclusive rather than definitive.

decisions. The principles and guidelines address how the system is constructed, and how the system is expected to change or evolve.

2.3 Architecture-Based Development

Software architecture is critical to the successful development and maintenance of large and complex software systems. Booch, in his book *Object-Oriented Design with Applications*, states the following:

We have observed two traits common to virtually all of the successful object-oriented systems we have encountered, and noticeably absent from the ones we count as failures: the existence of a strong architectural vision and the application of a well managed iterative and incremental development cycle. [Booch 91]

In our opinion, the applicability of this observation extends to software systems in general, and more than hints at the value of architecture-based development.

Architecture-based development is a process that utilizes the software architecture as the primary tool for the design, evolution, implementation, management, migration, and understanding of a software system. It involves [Clements 96]

- organizing the work products around the architecture
- implementing the software system based on the architecture
- maintaining the implementation to reflect changes in, and to ensure conformance to, the architecture

Architecture-based development includes the normal design, program, and test activities. It also has the following characteristics [Clements 96]:

- A domain analysis identifies and takes advantage of the commonalities and variations within a specific domain through an understanding of the underlying domain requirements. The belief is that a more general set of requirements will yield a more flexible, robust software system.
- Iterative selection or development of a software architecture involves prototyping, testing, measuring, and analyzing the software architecture [Kruchten 95]. This provides early assurance that the system will perform as required. Further, it has been argued that the iterative development of a software architecture by a small team yields conceptual integrity [Brooks 95].
- Effective representation and communication of the architecture to the stakeholders provides a common understanding of the software system and supports the coordinated effort required for system development.
- Evaluation and verification of the software architecture confirms that the selected or developed architecture can satisfy the system requirements

before the system is implemented. This is strongly tied to the selection or development of an architecture.

- Enforced system conformance to the selected architecture during system development ensures the conceptual integrity of the system. We would extend this to include system maintenance. Without enforcement, the decisions of the system designers can be lost or violated, causing the software architecture to fail as a basis for effective communication among the system stakeholders.

The benefits that can be expected from architecture-based development include [Garlan 95]

- understandability, by exposing the high-level constraints on system design and the rationale for specific architectural choices
- potential reuse of both the software components and the architectural framework, possibly forming the basis for a product line
- system evolution, by making the direction of expected system evolution explicit, improving the understanding and predictability for modification (cost and schedule), and separating the component functionality from the coordination of the components
- analysis, by the consistency of and conformance to a particular architectural style
- management, by using the definition of the software architecture as a key milestone, and supporting scheduling and cost estimation of development and maintenance work

3 Aircrew Trainer Simulation Software

3.1 Introduction

The goal of this chapter is to provide a high-level description of the simulation software required by aircrew trainer systems. We describe

- aircrew trainers and their associated benefits
- the role of simulation software in a trainer
- the traditional data-driven architecture of aircrew trainer simulation software
- the concerns of the B-2 trainer software team at the start of the development effort

Finally we provide a brief analysis of the inherent difficulties associated with data-driven architectures.

3.2 Aircrew Trainers

3.2.1 Overview

Simulation systems, such as aircrew trainers, mimic the behavior of a real-world system in a meaningful manner. By meaningful, me mean that the simulation system has an intended purpose and a known level of correlation and accuracy with the real-world system (i.e., fidelity). An aircrew trainer is a simulation system that reproduces the behavior of an aircraft and its operating environment (e.g., weather conditions) for the training of an aircrew (the pilot, copilot, navigator, etc.). A wide variety of aircrew trainers exists. In this report, the aircrew trainers that we refer to are the high-fidelity aircrew trainers currently used by the U. S. Air Force.

Figure 1 provides an overview of the operation of a generic aircrew trainer. The boxes represent the systems of the trainer, while the thick arrows indicate the flow of information. The simulation software of particular interest to this case study is located in the dotted rectangle titled “Simulation Systems.” The air vehicle system is responsible for calculating the behavior of the simulated aircraft, while the environment system is responsible for calculating the behavior of the tactical and natural environment of the simulated aircraft.

The trainer operates as a classic feedback loop. Based on the information presented to the crew by the cueing systems, the cockpit controls can alter the state of the trainer. The altered state is then presented to the crew via the cueing systems. Additional control of the trainer is provided by the instructor operator station (IOS), allowing the instructor to monitor and alter the state of the trainer. The IOS allows the instructor to tailor specific missions and situations for the training of the aircrew.

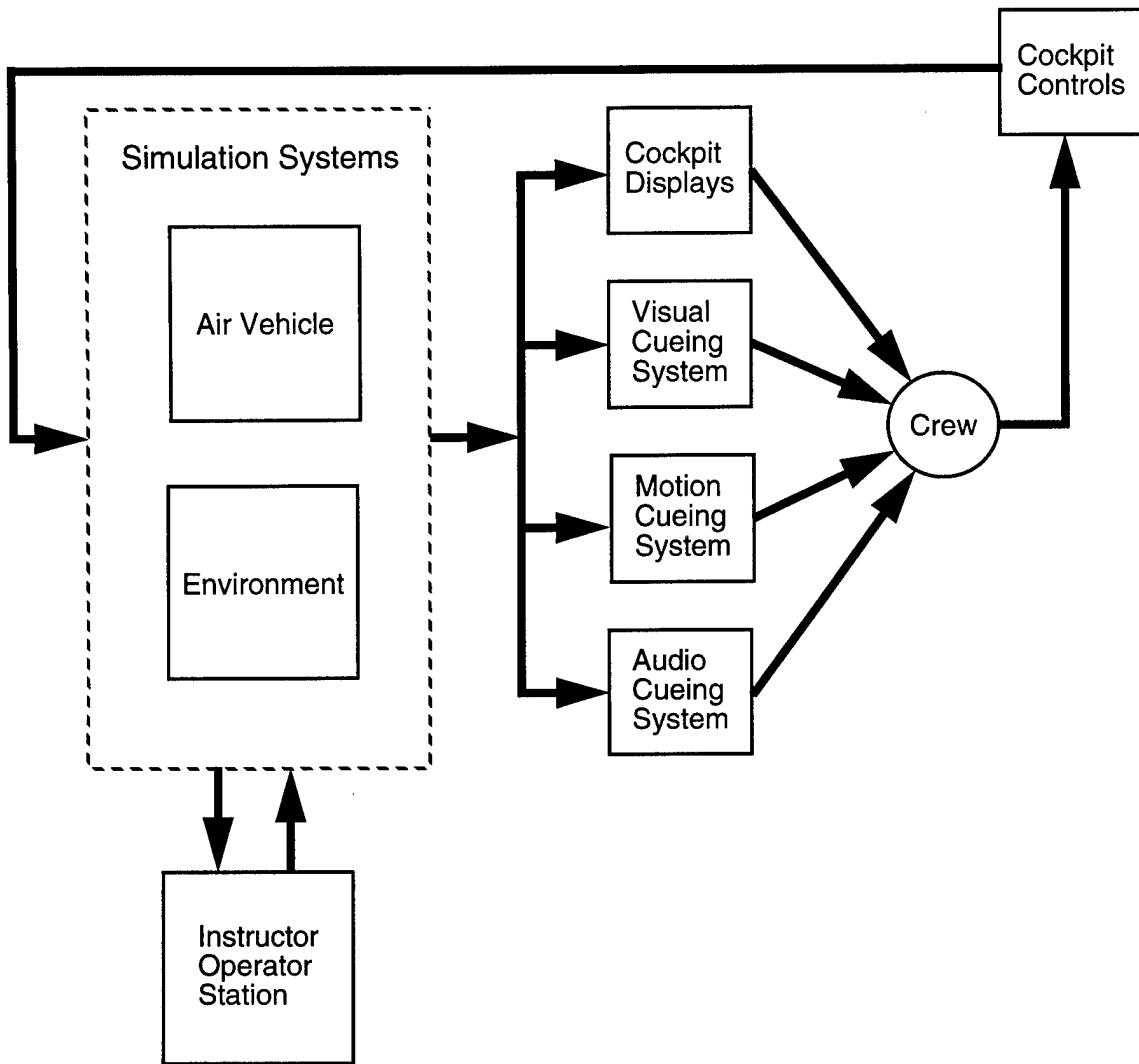


Figure 1: Operation of an AircREW Trainer

The cockpit of the aircREW trainer is designed to realistically reproduce the cockpit of the simulated aircrew, both in appearance and behavior. The trainer includes a variety of cueing systems that provide audio, visual, and motion cues to the crew. These systems employ physical devices to simulate turning, climbing, etc. The cockpit controls, in turn, respond to inputs from the crew.

3.2.2 Associated Benefits

Creating a high-fidelity aircREW trainer is no small enterprise. The B-2 project, for example, has employed 250 engineers for 7 years, and has produced 1.7 million lines of Ada code for the simulator. The simulation software, however, represents only a part of the B-2 trainer (e.g., the physical cockpit of the trainer, the cueing system's hardware and display software).

However, there are many compelling benefits associated with the use of trainers, including improved cost, safety, flexibility, convenience, and security, as explained below:

- cost: Modern aircraft are expensive to build, operate, and maintain. Single use weapons can also be prohibitively expensive for training purposes [Rolle 86].
- safety: The trainer can provide experience with high-risk situations without endangering the crew or a plane.
- flexibility and convenience: Training should include interactions with potential enemy aircraft and weaponry, as well as unusual missions and situations (e.g., severe weather conditions). Trainers can provide an aircrew with such experiences; further, these missions can be used to assess an aircrew's skills and repeated to fine-tune the aircrew's performance.
- security: Trainers can minimize exposure of secret aircraft, weaponry, and capabilities while providing the aircrew with necessary training.

3.3 Aircrew Trainer Simulation Software

Our primary focus in this paper is on the software required by an aircrew trainer to simulate the aircraft (shown as the dotted box titled "Simulation Systems" in Figure 1). This section explains the operation of the simulation software in the aircrew trainer.

The fundamental problem addressed by the trainer's simulation software is to calculate how the aircraft will behave at any given time. The aircrew trainer simulation software is designed based on a conceptual model of the aircraft and its operating environment. A model is an abstraction of the real world that bridges the gap between the aircraft and the simulation software. The model identifies key objects from the aircraft and operating environment, and captures the behavior of those objects and the relationships between them.⁴ The model also captures more global characteristics of the aircraft that are not directly reflected in the objects or their behavior: characteristics include the aircraft's dynamic nature (e.g., parts of an aircraft being upgraded, changes in the on-board computer hardware or software, and new navigational systems). A model addresses the primary characteristics of a simulation system; i.e., that it *mimics* the behavior of a real-world system in a *meaningful manner*.

In the model, an object's behavior at any given time is characterized by, and depends upon, the state of the object, and possibly the state of other objects in the model. State associates a particular time-dependent behavior with an object. For example, consider an aircraft's jet engine as an object in a particular model of an aircraft. The engine will behave differently depending on whether it is on or off. "On" and "off" are states of the engine. Further, the state of that engine can change from "on" to "off" depending on the state of other objects in the model (the fuel tank or fuel pump, for example).

4. Although the elements of a model are referred to as objects, there is no assumption at this point of an object-based or object-oriented design.

The model forms the basis for the design of the simulation software. The trainer's simulation software calculates the state of the simulated aircraft by calculating the state of each of the key objects identified by the model. Each object from the model has associated code (i.e., a state-calculating component) in the software system that computes that object's state. Those components are then coordinated into a solution to the original problem: how to compute the state of the aircraft.

There are complications to this simplistic view of the simulator software. The state of the aircraft is calculated discretely rather than continuously; that is, state is updated at the end of each of a series of discrete time periods, based on the state at the start of the period. As long as the time periods are short enough, the behavior of the simulated aircraft can be effectively mimicked. The length of the time periods depends, in part, on the fidelity required by the simulation system.

The state calculations in a discrete simulation system must be managed; this management includes the following:

- The state-calculating components must be scheduled to execute at the proper times within those discrete time periods.
- Since the state of an object can depend on the state of other objects, state information must be communicated between the software components.
- Since the simulated aircraft's systems operate at various rates, their associated software components must execute at corresponding rates to faithfully reproduce the simulated aircraft's behavior.

Finally, the trainer incurs overhead while managing this scheduling and communication.

Figure 2 illustrates the simulator processing that occurs during a typical time period.

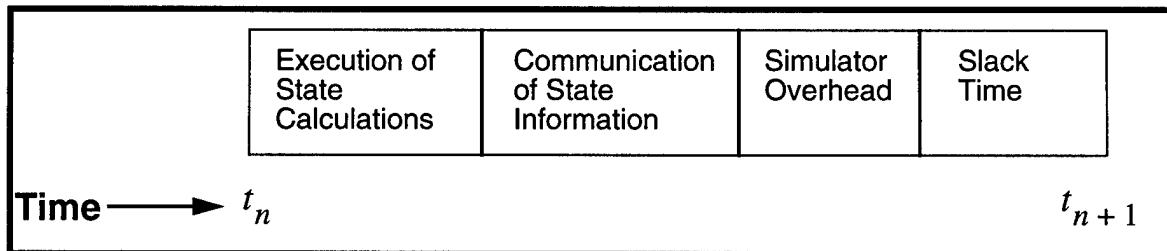


Figure 2: Simulator Time Period

The timing of the state information update is critical. The state calculations from Figure 2 require the state information as it exists at the start of the time period. If some part of the state information is updated during the time period, and subsequently used during that time period, the results of the calculation will be erroneous. This is called the data coherence problem.

There are many different possible software architectures that could support the simulation software in an aircrew trainer. In the following sections, we will discuss an early family of software architectures for aircrew trainers.

3.4 Data-Driven Software Architecture

3.4.1 Overview

Before the advent of structural modeling, the data-driven software architecture was the de facto standard for aircrew trainer simulation software. Dating back to the earliest digital computer-based aircrew trainers, data-driven software architectures provided a good solution to the trainer simulation software problem as it existed in the 60s and 70s [Rolle 86]. Data-driven architectures have been successfully used to simulate aircraft as complex as the B-52.

Figure 3 illustrates the high-level interactions of a typical simulator based on the data-driven architecture. The thin arrows represent data flow, while the thick arrow represents flow of control. The thin rectangles represent data areas, while the thick rectangles represent code.

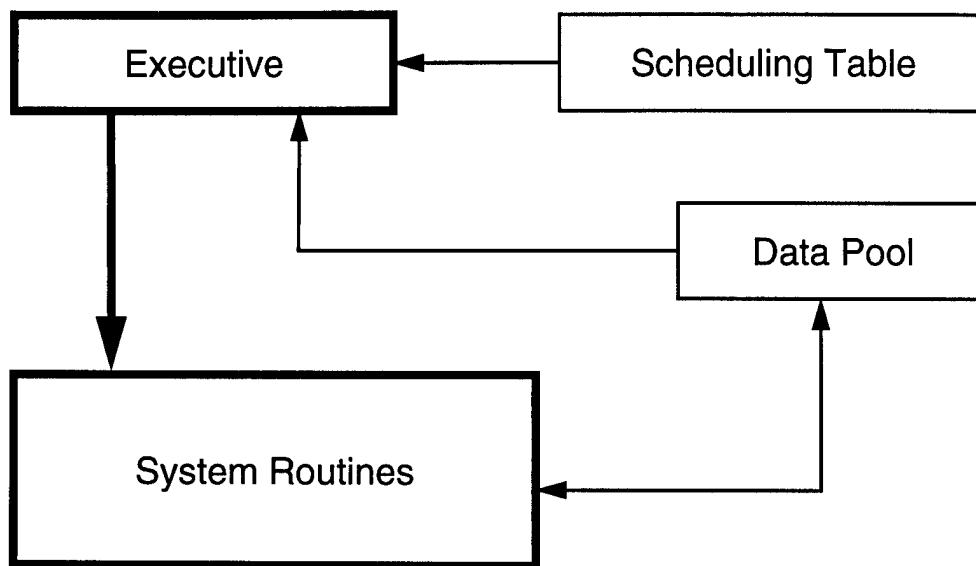


Figure 3: Data-Driven Software Architecture

The data-driven architecture consists of the

- executive: code responsible for scheduling the system routines based on the contents of the scheduling table
- system routines: code responsible for calculating the state of the trainer
- scheduling table: data specifying the scheduling requirements for the system routines
- data pool: data area shared by the system routines for storage and communication of state information

The executive dispatches the various system routines based on the entries from the scheduling table and information from the data pool. Each system routine calculates state information based on the contents of the data pool, and then updates the data pool to share those results with other system routines.

The system routines are a set of software subsystems that correspond to the major functions of the simulated aircraft. Examples of these system routines could include, for example, taxiing and landing. The physical structure of the aircraft cuts across the functional subsystems. For example, the landing gear is intimately involved in both taxiing and landing. The state of the landing gear must be considered when calculating the state of both the taxiing and landing subsystems. The data-driven architecture views the subsystems as black boxes, mapping inputs to outputs, but hiding the subsystem's internal behavior. This forces both the taxiing and landing subsystems to contain state calculations for the landing gear.

3.4.2 Difficulties with the Data-Driven Architecture

Data-driven architectures were an effective solution to the trainer software problem as it existed 20 years ago; the problem, however, has changed dramatically. Today's airplanes are more complex and dynamic; today's computer hardware is less expensive and more powerful, while today's software is more complex and expensive. Application of the data-driven architecture to software trainers for modern aircraft is something of a mismatch between the newer problem and the older solution.

Symptoms of this mismatch surface during integration and maintenance of systems based on the data-driven architecture. The integration of these systems has been described as something of a "big bang." Most, if not all, of the trainer simulation software has to be written before integration testing can begin. There is also the concern that once the system is integrated it will not perform as required, since system testing cannot begin until after integration.

Maintenance of such systems is also difficult. The functional decomposition of the data-driven architecture spreads the state information and calculations, as well as the communication of state information, across the subsystems. Data coherence problems have resulted from this spreading. Further, development of different subsystems by different software teams leads to multiple mechanisms for representation, calculation, and communication of state information. Hence, state calculations are difficult to locate; once located, it is difficult to determine the effects of a change.

Maintenance is further complicated by communication difficulties between the domain experts and the software experts. The domain experts understand the physical structure of the aircraft; however, the software structure understood by the software experts reflects the functional, rather than the physical, structure of the aircraft. The functional decomposition fails to provide a direct mapping between the physical structure of the aircraft and the software. In effect, the domain and software experts have a different vocabulary.

The spreading of state representation, calculation, and communication leads to interdependencies among the data-driven architecture's system routines. Those interdependencies complicate the

- concurrent development of the simulator software
- incremental development of the simulator software
- simulation of malfunctions⁵
- freezing, saving, and restoring of state information

3.5 The B-2 Project

As the simulated aircraft became more complex, the software-related difficulties with the trainers based on the data-driven architecture became more severe. The B-2 program is a particularly relevant example, since the initial software architecture for the B-2 trainer was data driven, but its current architecture is based on structural modeling. There were several concerns at the beginning of the B-2 trainer effort that led to the development of structural modeling. The B-2 was probably not the first trainer where these concerns surfaced, but it was the first program to address them by using structural modeling.

The B-2 was developed as a black program: the existence of the plane was highly classified. Development of the trainer began in the mid 1980s. Like several previous Air Force trainers, the B-2 trainer would be implemented as a distributed simulation with hard real-time requirements. Unlike earlier trainers, the B-2 trainer would use Ada as the primary implementation language.

From the start of the effort, the B-2 trainer developers knew that the

- trainer would be long-lived: At the time the trainer work began, the B-2 itself, and hence the trainer, was projected to have a lifetime of 30 years.
- simulation software system would be large: Based on the complexity of the B-2 and previous experience with constructing aircrew trainers, the developers believed that the B-2 trainer would require a lot of software (in fact, 1.7 million lines of Ada code were written) and that the integration of that software would be very difficult.
- development effort would be distributed: They also knew that different sections of the trainer software would be developed at different, geographically distant locations, and those sections would become available at different times. For example, the radar-system simulation software was to be developed on the west coast, while the rest of the simulation software was

5. A malfunction is, in effect, a sudden change in a portion of the simulator's state information. For example, a hydraulic pump's output can suddenly drop to zero.

to be developed on the east coast. It was also known that the radar system would be added late in the development.

- initial requirements were unknowable: When work began on the trainer, the B-2 was not yet built; a full description of B-2 did not exist. In short, the aircraft and the trainer would be built in parallel.
- trainer would be dynamic: Changes to the B-2's systems, such as new weapons systems, were expected over the lifetime of the aircraft and would have to be reflected in the trainer.

Long-lived systems imply that ease of maintenance and modifiability are more of a concern, while larger systems imply that the integration of the software is more of a concern. The distributed development required by the B-2 trainer implied the need for unambiguous communication between developers and coordination of their efforts. The initially unknowable requirements, and the continuing dynamic nature of the B-2 implied that the simulation software would need to evolve with the aircraft from the start of the trainer development effort.

3.6 Summary

We view the difficulties and concerns expressed in the previous sections of this chapter as symptoms of the lack of changeability of both the data-driven architecture and the development process it supports. Changeability is defined as the ease with which a software system can be changed throughout its life cycle [Clements 96]. This definition extends the standard notions of modifiability and maintainability to include changes in requirements and specifications.

4 Structural Modeling

4.1 Introduction

The goal of this chapter is to explain the use of structural modeling in the development of air vehicle simulation software. We describe

- the partitioning of the components in the software architecture based on structural modeling
- structural types and structural models, and their relationship to the air vehicle software architecture
- the development artifacts associated with structural models

Structural modeling is based on the realization that the difficulties and concerns expressed in the previous chapter represent design issues to be addressed by the software architecture. It can be understood as a design effort to produce a software architecture that supports changeability. The general software design principles to be applied include

- separation of domain commonality from variability: separating the code expected to remain constant for the given domain from the code expected to change
- object-based partitioning based on the physical structure of the simulated aircraft
- separation of concerns
- restriction of interfaces and mechanisms for communication between components
- restriction of the flow of control

An architectural style “defines constraints on the form and structure of a family of architectural instances” [Garlan 95]. This chapter explains structural modeling as an architectural style that supports changeability in aircrew trainer simulation software by restricting the structure, behavior, and organization of the components of the software architecture. The process involves partitioning the system (ultimately into the components of the architecture), and then restricting the coordination of and communication between the resultant partitions.

The simulation software architecture based on structural modeling has evolved over the past decade, particularly in the area of communication. Although prototype versions of the IOS and synthetic environment based on structural modeling exist (see Figure 1), only the air vehicle is both based on structural modeling and currently in use in operational aircrew trainers. In this chapter, we will examine the architecture of the air vehicle as presented in the Guidebook [AS-CYW 94] and discuss the following:

- separation of the general simulator support software from the aircraft-specific software

- object-based decomposition of the aircraft-specific software
- use of structural types to capture common functionality between components and to restrict the structure of those components
- use of a structural model to generate a software architecture from a set of structural types
- the development artifacts used to create and support the software architecture

4.2 Separation of Domain Commonality from Variability

At the highest design level, the air vehicle simulation software is partitioned by function, into general simulator support and aircraft-specific state calculation software (see Figure 4). The simulator support software, traditionally referred to as the executive, handles the scheduling and synchronization of the air vehicle simulation, and the internal and external communication of state information. The aircraft-specific software calculates the state information for the simulated aircraft.

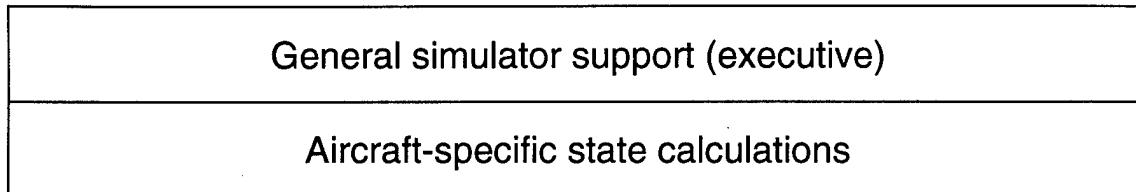


Figure 4: High-Level Partitioning of Simulator Software

This partitioning separates the software that can change from aircraft to aircraft from the software that is constant for the domain (i.e., aircraft trainers). Hence the executive can be aircraft independent (i.e., insulated from changes in the simulated aircraft). It can provide a basis for an architectural framework for the

- incremental development of a trainer for a specific aircraft
- development of other trainer simulation systems (i.e., for different aircraft).

Note also that this partitioning isolates the software corresponding to the user's view of the trainer (the aircraft-specific calculations). This is a first step towards improving communication between the software experts and the domain experts.

A more detailed examination of the executive is presented in Appendix 1. Interested readers should refer to the appendix after reading the rest of the current chapter.

4.3 Object-Based Components

The aircraft-specific state calculation software is partitioned using an object-based strategy that reflects the least common denominator of the physical structure of the aircraft: i.e., the part. The behavior of the simulated aircraft results from the interactions and relationships be-

tween the parts of the aircraft; the object-based partitioning is the first step towards mirroring this in the software. Figure 5 shows an example of this partitioning for some of the parts in a hydraulic system of the aircraft.

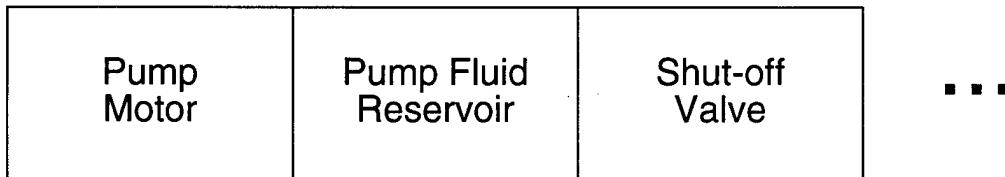


Figure 5: Partitioning of the Aircraft-Specific State Calculations

The part is the smallest expected unit of change of the simulated aircraft; significant change affecting the simulation of an aircraft is not expected to occur at a lower level in the physical aircraft. Hence this partitioning is a first step towards supporting modifiability. As a result of the object-based partitioning, each state-calculating component encapsulates its own portion of the overall aircraft state and its related calculation, limiting the effects of changes to that information and calculation. This further promotes independent development. Finally, the view of the airplane as a set of cooperating parts is consistent with the user's view of the aircraft, further improving communication between the software expert and the user.

Every state-calculating component is responsible only for calculating the state information for a part of the simulated aircraft (e.g., a fuel tank). Responsibility for communicating state information to and from the state-calculating component, as well as scheduling the execution of that component, is external to the component. Hence each state-calculating component requires interfaces that cause the component to

- be executed at the proper time
- receive required state information from other state-calculating components
- supply its state information to other state-calculating components
- reset its own state (e.g., instructor restarts the exercise)
- alter its own state information aperiodically (e.g., an instructor-initiated malfunction).

These commonalities of general functionality (in this case, to calculate the state of a particular aircraft part) and resultant interfaces can be exploited by restricting all the state-calculating components to a common set of visible operations. For example, every state-calculating component can be required to have exactly three visible operations: *update*, *configure*, and *process_event*, where

- *Update* calculates the next state of its associated part based on its current state and its input parameters which contain the required state information from other state-calculating components. *Update* in turn makes its component's state information available in its output parameters.

- *Configure* resets the internal state of the component based on its input parameters.
- *Process_event* alters the internal state of the component based on its input parameters (e.g., malfunctions).

4.4 Structural Types

The restriction, or uniform treatment, of the state-calculating components simplifies this construction and promotes understandability. Structurally, every state-calculating component is the same. Each such component has clearly defined responsibilities to the rest of the system. State information enters and leaves every such component by the same mechanism.

The pairing of the general functionality of a related group of components with an associated restricted set of visible operations yields a structural type. A structural type captures the commonalities of form and interaction of a group of architectural components, specifying the common type of functionality and the set of capabilities (i.e., operations) shared by the members of that group.⁶ The variability (i.e., the precise nature of the state information and its calculation) is relegated to the parameter lists and bodies of the operations, respectively.

A structural type is traditionally represented as a box (or icon), with its visible operations highlighted. For example, the structural type for the state-calculating components [ASCYW 94] is shown in Figure 6.



Figure 6: Component Structural Type

4.5 Subsystems

The parts of the simulated aircraft group naturally into the physical subsystems of the aircraft (e.g., the hydraulic braking subsystem), and function in concert to physically affect the behavior of the aircraft. This grouping is mirrored in the architecture: the subsystem controller is a component that groups and controls logically related state-calculating components.⁷

6. This is a high-level notion of functionality: in this case, to calculate the state associated with a part in the simulated aircraft. It ignores lower level functionality. Hence, other commonalities across state-calculating compounds are not captured. For example, all the pumps in the simulated aircraft push fluid through a system.
7. The grouping in the air vehicle is based on the physical subsystems of the simulated aircraft; however, the grouping can, for example, be rate based to enhance performance. For convenience, we define a subsystem to be a subsystem controller together with its associated state-calculating components.

The state-calculating component has exactly one view of the rest of the air vehicle software system, as realized by its parameter list. The subsystem controller, however, has two views of the rest of the system: outside of the subsystem and within the subsystem.

4.5.1 The Subsystem from the Outside

From outside of the subsystem, the air vehicle simulation software consists of the executive and various subsystems (Figure 7). In a sense, the subsystem bridges the gap between the general support and aircraft-specific code.

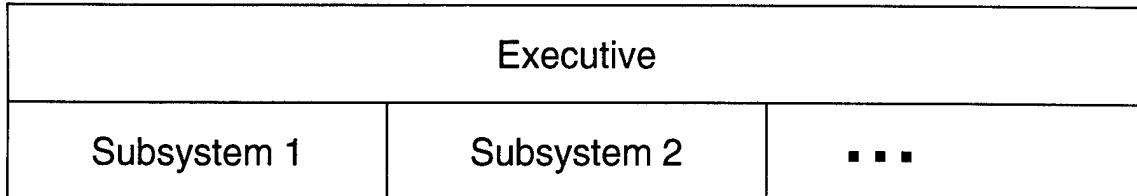


Figure 7: View Outside of the Subsystem

In the same way that the state-calculating component encapsulates the state information and calculation for an associated part of the simulated aircraft, the subsystem encapsulates the state information and calculation for a group of state-calculating components. Responsibility for communicating state information to and from the subsystem, as well as scheduling the execution of the subsystem, is external to the subsystem. Hence, each subsystem controller requires mechanisms that cause the subsystem to

- execute at the proper time
- receive state information from other subsystems
- supply its state information to other subsystems
- reset its state (e.g., instructor restarts the exercise)
- alter its state information aperiodically (e.g., an instructor-initiated malfunction)

Just as with the state-calculating components, visible operations are defined for the subsystem controller:

- *Update* calculates the next state of the subsystem based on the subsystem's current state and input parameters.
- *Configure* resets the internal state of the subsystem based on its input parameters.
- *Process_event* alters the internal state of the subsystem based on its input parameters.
- *Import* causes the subsystem controller to access the export areas of the other system controllers.

The *update* operation for the subsystem controller is different from the *update* for a state-calculating component. The state information required by, or supplied by a subsystem, is not passed by parameters;⁸ rather, state information is communicated between subsystems by use of single-writer, multiple-readers export areas, typically implemented by shared memory.

Controlled accessing of these export areas is required to prevent data cohesion problems. This control is provided locally by the subsystem controller's *import* operation, and is exercised externally by the executive.⁹

A subsystem controller structural type can be defined using an argument similar to the one used with the state-calculating components (see Figure 8).

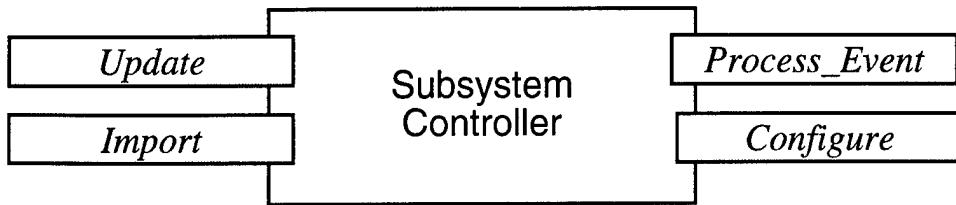


Figure 8: Subsystem Controller Structural Type

From the outside of the subsystem, the subsystem controller encapsulates the subsystem. This encapsulation provides a common abstraction or view of the aircraft-specific code to the executive, hides the existence of the state-calculating components from the rest of the system, and allows the executive to schedule subsystems. This encapsulation defers the scheduling of the state-calculating components to the subsystem controller, and simplifies the executive's scheduling tables.

4.5.2 The Subsystem from the Inside

From within the subsystem, the subsystem controller uses the visible operations of its state-calculating components to

- schedule and invoke its components
- supply state information to its components
- receive state information from its components
- reset the state of its components (e.g., instructor restarts the exercise)
- alter the state information of its components aperiodically (e.g., an instructor-initiated malfunction)

⁸. The use of parameters would require executive knowledge of aircraft-specific state information, violating the original partitioning (i.e., general support vs. aircraft-specific code).

⁹. The import operation has been eliminated from later architectures, as the data cohesion problem is, for the most part, addressed by the state-calculating component's encapsulation of its state information.

State information imported into a subsystem or exported from a subsystem is maintained in the subsystem controller as local variables.

From the inside of a subsystem, the subsystem controller

- hides the origin of supplied state information from the state-calculating components: Each state-calculating component sees only its own state information and the input parameters supplied by the subsystem controller. The input parameters contain the state information required by a state-calculating component, but owned by a different state-calculating component.
- schedules and invokes its state calculating components. This simplifies the scheduling tables, further supporting the abstraction of aircraft-specific knowledge from the executive.
- provides a uniform mechanism for adding and deleting state-calculating components, supporting modifiability by limiting the effects of additions and deletions to the subsystem.
- limits the effects of changes to existing state-calculating components. Only changes to the communicated state of a state-calculating component can potentially affect the subsystem controller and other state-calculating components.

The subsystem grouping is understood by the users of the aircrew trainer. Grouping the state-calculating components based on the physical subsystems of the simulated aircraft further promotes communication between the domain and software experts.

4.6 Structural Models

In the previous sections, we presented the partitioning of the air vehicle as a set of structural types. A particular software architecture for the air vehicle simulation consists of instances of the previously described structural types; those instances are organized within the architecture according to a set of rules governing their interactions. A structural model is a small collection of structural types, together with the rules for organizing instances of the structural types into an architecture [ASCYW 94].

A structural model can be thought of as a set of restrictions that constrains

- the components of the architecture to particular forms and interactions based on a system-wide partitioning (i.e., into the object-based state-calculating components, subsystem controllers, etc., from the previous sections)
- each functional type of component (i.e., structural type) to a common set of capabilities (i.e., operations for interacting and communicating with the other components of the architecture)
- the organization within the architecture according to rules that govern the grouping of, and the flow of control between, the components of the architecture (Appendix B contains more details on the flow of control.)

4.7 Development Artifacts

Three artifacts are used to develop the air vehicle simulation software: specification forms, software templates, and an integration harness. Specification forms document the design specifications for the subsystem controllers and state-calculating components. A subsystem specification form contains [ASCYW 94]:

- subsystem identification
- subsystem description
- data interfaces
- design assumptions
- temporal dependencies
- event algorithms
- cyclic algorithms
- subsystem components
- resource allocation
- reference data
- design concerns
- requirements traceability

A similar specification form is used for state-calculating components.

A software template is an Ada 83 code skeleton that acts as a bridge between the specification form and the Ada source code for a subsystem controller or state-calculating component. There is a direct correlation between the specification forms and the software templates. The templates and specification forms simplify the coding process and support a uniform style of coding within the air vehicle.

The integration harness is a skeleton version of the simulation system that consists of the executive, subsystem controllers, and synthetic workloads for the anticipated state-calculating components. It provides an initial “operational model of the simulation software architecture [ASCYW 94].” The integration harness can be used to support the incremental development of the air vehicle implementation by merging components into the harness as they are completed. Further, the integration harness provides a method for early (and continuing) verification of the architecture’s performance.

4.8 Summary

Structural modeling

- defines a small set of structural types that restrict the components of the simulator software architecture; that is, it restricts the responsibilities of, and the interactions and flow of control between, the software components.
- adopts a uniform, controlled, and explicit mechanism for the communication of state information
- adopts a finer granularity for the state-calculating software components than that of the data-driven architecture (i.e., at the aircraft part rather than the functional subsystem level)
- encapsulates the state information calculated by a software component in that component (i.e., a state-calculating component owns the state it calculates)
- encapsulates the subsystem-level state information and communication within the subsystem

The architecture generated by structural modeling exhibits abstraction, encapsulation, modularity, and typing.¹⁰ The components are self contained and predictably replaceable. The effects of change are localized, relatively easily understood, and predictable. The connections between the software components and their subsystem controller are explicit, consistent, and exposed, as are the other connections within the air vehicle architecture. Further, the semantics behind the connections are consistent.

10. The relationship of structural modeling to current object-oriented design techniques will be discussed in Chapter 6.

5 Organizational Factors

5.1 Introduction

This chapter describes the impact of structural modeling on the organizations interviewed. Specifically, we describe the impact on the interviewed organization's

- management structure
- staffing profiles
- development approach, including life cycle, methods and tools, and documentation
- management of the learning curve for structural modeling

Structural modeling was developed in the mid-80s through the efforts of key members of the engineering office at ASC/YW. Although the initial incentive was provided by ASC/YW, many senior software engineers from various aircrew trainer projects and the SEI worked cooperatively to develop and mature structural modeling through its use with Air Force aircrew trainers. Funding was provided by ASC/YW for the development of the approach, training, and consulting to assist individual projects.

While all aircrew trainers are the responsibility of ASC/YW, each aircrew trainer was developed for, and administered by, a separate program office (for example, the B-2). Further, each trainer was built by a different contractor. This situation made ASC/YW's enforcement of a technical solution difficult.

Structural modeling was introduced into the development of new simulators at the same time as several other emerging software development technologies and processes, namely, Ada83 and its development environments, and DoD-STD-2167 (and later DoD-STD-2167A) for defense system software development.¹¹ DoD-STD-2167 was intended to facilitate the construction and maintenance of large, defense, software-intensive systems. Ada83 can enable the development of higher quality, more maintainable systems.

¹¹ DoD-STD-2167 requires a rigorous set of documents, milestones, audits, and reviews tailored to the needs of the specific project. Typical interpretation of the standard was the use of extensive documentation and of a waterfall life cycle. DoD-STD-2167 was issued in the mid-80s, roughly as the B-2 project started. DoD-STD-2167A was released in 1988 and used by the subsequent flight simulators. To improve the readability of this report, we have used DoD-STD-2167 to refer to the series of DoD-STD-2167s.

The following projects have used structural modeling with SEI support:

- B-2 Weapons System Trainer, started in 1986, with 1.7 million lines of Ada code developed by Hughes Training/Link Division. The system is operational. Hughes/Link continues to maintain the system.
- C-17 Aircrew Training System (ATS), started in 1990, with 350,000 lines of Ada code developed by McDonnell Douglas. The system is operational. McDonnell Douglas owns the software and performs the training for the Air Force.
- Special Operations Forces (SOF) ATS, supporting the C-130, started in 1991, with 750,000 lines of Ada code developed by Loral Federal Systems. One-half of the system uses structural modeling. Due to budget shifts, the system is still under development, although Loral has delivered partial functionality.
- Simulator Electric Combat Training (SECT), started in 1992, with 250,000 lines of Ada code developed by AAI Corporation. Three-quarters of the system uses structural modeling. The project is currently in the implementation phase.

Over the course of our interviews, we spoke with contractors who used structural modeling in its infancy, as well as contractors who used the technology after maturation and broader acceptance within the flight-simulator community. We spent one day on site with seven members of the B-2 project. Extended phone interviews were conducted with senior technical members from the C-17 and SOF/ATS projects.

This chapter summarizes the results that the above projects reported and focuses on the impact of structural modeling on the contractor organizations. The SEI studied their organizational structures and staff profiles; the processes, methods, and tools that make up their software development approaches; and the learning curve for the various organizations.

5.2 Results

All the interviewed projects felt that structural modeling had greatly improved the overall quality of their simulation systems. Examples of reported improvements included

- significant reduction in test problems. In a previous data-driven simulator of comparable size (the B-52), 2000 - 3000 test descriptions (test problems) were identified during factory acceptance testing. With their structural modeling project, 600 - 700 test descriptions were reported. They found the problems easier to correct; many resulted from misunderstandings with the documentation.
- significant reduction in staff for installation. On-site staff during initial installation and use was reduced by about 50%. They found fault detection and correction significantly easier.
- significant reduction in test expense. Staff typically could isolate a reported problem offline rather than going to a site. Reduced testing and fault isolation on the actual trainer was particularly significant given the high expense of trainer time.¹²

¹². Trainers frequently require high availability levels. For example, the B-2 must operate at 95% availability. The Air Force cannot afford to have trainers used for fault isolation.

- reduction in side-effects from software changes. Most projects noted that side-effects from software changes were a rare occurrence, primarily due to the encapsulation of functionality in subsystems.
- extreme ease of integration. All projects commented on the lack of the “big-bang” effect when compared to their previous data-driven simulators of comparable size. The use of common structural types for components and a standard mechanism for communication between components were cited as key contributors.
- significant improvement in reuse. One project reported reuse across trainers of the software architecture, executive, and subsystem controllers.
- significantly reduced defects. Since the use of structural modeling, defect rates for one project are half that found on previous data-driven simulators. Figure 9 shows the cumulative results through integration for a particular project. (The dotted line is based on historical data, while the solid line is based on structural modeling data collected through integration.)

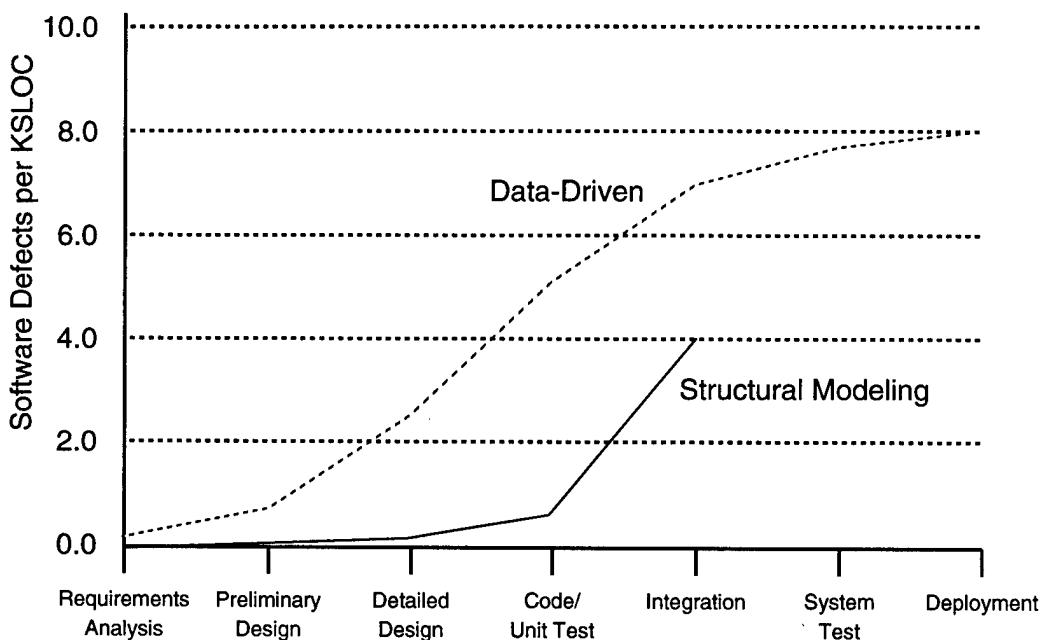


Figure 9: Reported Defects - Before and Since Using Structural Model

5.3 Impact on Organizational Structure

The contractor organizations for flight simulators include more than a software development group. They also include, at a minimum, a program-level, system engineering, and hardware engineering groups. All groups contribute to the eventual flight simulator, which is a system of software and hardware. Decisions made for either software or hardware have an impact on the total system by varying degrees. Structural modeling represented a significantly different approach to the software.

Structural modeling was introduced into the various simulator projects as primarily a software architecture and software construction technology. The software development groups were the first to use the technology and determine how to integrate it into their development operations. Other organizational parts of the projects, such as the program-level or system engineering groups, remained remote from the technology and did not change their organizational structures or procedures. All interviewed reported that this presented problems, which will be discussed later in this section.

To understand the changes faced by many of the software development groups, the organizational structures typically used prior to structural modeling must be understood. During the 1960s and 1970s, the flight simulator community used the data-driven software architecture. The organizational structures for the software development groups that we interviewed mirrored the functional structure of the data-driven architecture.

Several found this traditional software development organizational structure limiting. Glaize identified five areas of potential problems due to the mismatch of the software development group's organizational structure with the software development methodology and resulting architecture [Glaize 90]. The five potential problem areas noted were

- software ownership
- documentation ownership
- work breakdown structure and time accounting
- management
- logistics of geographically distributed development

All five problem areas are manifestations of the same mismatch between the organization and architecture structures.

For example, flight simulators provide simulations of take-offs and landings. With the data-driven simulator projects, one distinct part of the architecture handled take-offs and another handled landings. Both parts of the architecture would manipulate the landing gear. These projects typically assigned the take-offs part of the architecture to one functional group and the landings to another functional group. Each group would develop their own code to manipulate the landing gear. With this one-to-one mapping of architecture structure to functional groups, responsibility for software development, documentation development, work breakdown assignments, and geographically separate development were well defined.

In contrast, with an architecture created using structural modeling (based on objects), take-offs and landings are handled by several distinct parts of the architecture, with one part manipulating the landing gear. If the take-offs functional group and landings functional group are retained, responsibility for the software development, documentation, or work breakdown for the landing gear part of the architecture is unclear. As a result of this mismatch, a number of the organizations interviewed began modifying their software development organization to more closely match the software architecture developed with structural modeling.

The B-2 project initially used a large team of designers; this resulted in problems in gaining consensus and apportioning work. Since the B-2 project, most organizations have used a small team (five or six) of senior, experienced architects and senior designers to create the software architecture. Projects reported that this approach was more effective in creating a stable, consistent architecture that was ready for further detailed design and implementation by larger numbers of engineers.

Most projects reported that members of the architecture group typically were reassigned to other jobs once the software architecture was defined. Because of this, the architecture was sometimes compromised when issues encountered during implementation and maintenance were resolved; staff knowledgeable about the architecture, key design decisions, and underlying assumptions were no longer available to the project. Several postulated the need for a small, permanent architecture team with the authority and responsibility for the software architecture. This team would not only create a software architecture for a project, but would also control the evolution of the software architecture and resulting system.

5.4 Impact on Staffing Profiles

This section focuses on the impact to the software development groups of the flight simulator projects and highlights issues encountered between the software groups and other parts of the project. Staffing profiles were reported as unchanged outside the software development groups.

The knowledge and skills associated with each technology varied by role across a project. Software architects and designers, who were responsible for the creation of the architecture and the design of the subsystems, needed an in-depth knowledge of structural modeling, object-based design, and a working knowledge of Ada83. They needed the ability to look broadly at problems and identify potential patterns of commonality in the software architecture.

Software developers, who were responsible for implementation of the designs, needed a working knowledge of structural modeling, an in-depth knowledge of Ada83, their development tools, and a high-level knowledge of DoD-STD-2167 requirements.

Senior managers within the software development organization and at the program level had a limited knowledge of the engineering technologies used on the early structural modeling projects. Some mid-level managers and most low-level managers in the software development organization had a working knowledge of structural modeling, Ada83, their development tools, and DoD-STD-2167 requirements. All of those interviewed recommended expanding the awareness of mid and senior management both within the software organization and at the program level to help resolve staffing issues associated with the new technologies.

Early structural modeling projects reported that allocation of staff resources for each software development phase was based on previous data-driven simulators. Using these resource estimates, staff was added quickly during the early life-cycle phases. Several problems arose.

First, additional time was required to create and stabilize the software architecture. Second, adaptations to the software development processes to incorporate the new technologies were underway in parallel with the architecture definition. So, as changes were made in the processes or the architecture, more software developers were affected. With subsequent projects, resource allocations profiles were based on the growing experience using structural modeling. Projects tended to keep the number of software engineers low until the architecture was defined.

5.5 Impact on Development Approach

This section describes the effect of structural modeling on the project life cycle, methods and tools used, and project documentation.

5.5.1 Life Cycle

In the transition of the flight simulator projects to structural modeling (and related technologies), their development milestones and schedules are viewed from two levels: at the project-wide level, including non-air vehicle parts of the simulator, and at the software-development level for the air vehicle itself (where structural modeling was applied).

At the project-wide level, the traditional waterfall phases were used for software development (i.e., requirements analysis, preliminary design, detailed design, code and unit test, and integration defined in sequential order). Development milestones were contractually defined to coincide with the end of each phase. Program office reviews and documentation deliverables were tied to the completion of each phase. The estimated duration of each phase was based on previous data-driven flight simulator projects. Structural modeling has had little effect at the project-wide level.

Structural modeling had an immediate effect at the software-development level. On the initial structural modeling projects, the air vehicle software teams found they needed more time early in the life-cycle phases than contractually defined, but less time in test and integration. The additional up-front time was required to develop the structural types and architecture. Milestone and documentation review deadlines were difficult to meet. They found they needed a more evolutionary or iterative life cycle as a result of

- more software systems engineering, architecture definition, and prototyping to develop an effective architecture
- continual aircraft changes
- ongoing clarification of the required behavior of the simulator's end user

Over the course of the interviews, several projects reported the need to modify the project-wide (and contractually defined) life cycle to explicitly support the iterative or evolutionary nature of air vehicle software development using structural modeling.

5.5.2 Methods and Tools

All of those interviewed saw structural modeling as a process consisting of rules, procedures, and policies for designing and constructing systems. With the B-2 project, the structural types, code templates, and basic architecture for the air vehicle simulator were developed. All subsequent structural modeling projects reused the same structural types and basic architecture for the air vehicle software, including the same code templates for the structural types. Within the detailed design and code implementation of software components, each project applied its own processes, methods, and tools.

A large variability of processes, methods, and tools was reported. For example, one project developed their own requirements traceability tool that linked the customer's high-level requirements to their derived requirements and design documentation. Others used no comparable tool. One project invested heavily in creating its own tools to automate their enhanced software development process. Tool support included requirements traceability, configuration management, interface definition management, and development library management. The project emphatically noted that they would recommend projects avoid writing their own tools and procure commercially available tools.¹³

One project embarked on significantly reengineering its software development processes, methods, and tools in concert with structural modeling. For example, they

- developed an interface database that tracked the subsystem where the interface is defined and used, and the data element involved. To facilitate the management of the interfaces, they created a module connection language.
- created a configuration management capability to record problems, their analysis, changes identified, and tracking of the change process.
- incorporated code complexity measures, based on McCabe complexity measures, into their development process, and enforced these measures through management involvement and code reviews [Bedford 91].¹⁴ They believed that by managing the complexity of their code they could improve further the understandability, maintainability, and reliability of their software. Key to their success was the pragmatic application of complexity limits, allowing for the use of complex code when appropriate.
- expanded testing to include reliability measures. After researching the state of practice they adopted Musa's Basic Execution Time Model [Bedford 91]. The Musa Basic Execution Time Model provides estimates for a software unit's reliability (e.g., mean time to failure).

¹³. This project was classified; the high-security requirements made vendor support for hardware and software difficult. If hardware or software vendor support was required, the vendor had to supply staff with the same security classification as project personnel. Also, hardware or software could not be returned to the vendor for detailed problem isolation and resolution.

¹⁴. The McCabe Cyclomatic Complexity Measure indicates the relative complexity of a software unit based upon that unit's branching logic.

5.5.3 Documentation

Since all the structural modeling projects were following DoD-STD-2167, the project documentation, including that for the design, was required to conform to that standard. DoD-STD-2167 ties document requirements to the structure of a system, defining a system as a set of segments. Segments are decomposed into hardware computer configuration items (HWClIs) or software computer configuration items (CSClIs). CSClIs are decomposed into computer software components (CSCs). CSCs are optionally further decomposed into computer software units (CSUs). It is a project's responsibility to define a configuration item and its constituents in the hierarchy for their particular application.

All early projects reported that they had struggled with an appropriate mapping of DoD-STD-2167 components to their high-level architecture. Several projects had originally defined a CSCl as a subsystem. With the common structure of the subsystems, this resulted in large amounts of duplicative and redundant information. The CSCl was redefined to include the entire system for the air vehicle simulation software. The subsystem design then became internal to the CSCl. This decision drastically reduced the amount of documentation with no loss of information.

Customer requirements were typically supplied as English text of high-level operational capabilities. Senior engineers expanded and clarified the high-level customer requirements, producing a set of derived requirements that were captured in appropriate requirements documents for DoD-STD-2167. Most projects indicated that the high-level customer requirements rarely changed, but the derived requirements would change, potentially substantially, through detailed design and coding.

The resulting software architecture satisfied the derived requirements. The design documentation described the design process used, the resulting architecture, and constituent objects. English text and project-defined notations described the design. English text was used to indicate the scope of the simulation (what would or would not be simulated) and any assumptions. Some projects also used the aircraft diagrams with annotations capturing the scope of the simulation. For the early projects, the ASC/YW engineering office was involved in defining the scope and level of detail for the design documentation.

Most projects reported that design specifications used standard formats within their own project. Although there was no standard required across all Air Force flight simulator projects, many of the later projects have reused part of the design document from earlier projects. They reported that they reused the design document approach or outline, the description of the structural modeling approach, structural types and templates, and the description of the basic high-level architecture.

Examples of design specification formats used across the projects included

- textual descriptions of the objects in a subsystem and Ada83 package skeletons for subsystem controllers

- Ada83-PDL and math-model information added during detailed design (the amount of detail depended on the complexity of the math models)
- functionally oriented flow charts for preliminary design, and then a combination of Booch diagrams and structure diagrams (to capture control mechanisms) for detailed designs

Hughes/Link Division created its own notation [Bennett 92].

5.6 Managing the Learning Curve

For all projects, the shift to structural modeling coincided with a shift to Ada83 and DoD-STD-2167. The implementation language for previous systems was typically Fortran with some C or JOVIAL. Most projects were not accustomed to the additional control and structure required of the new technologies. Early projects had little or no experience with Ada83, DoD-STD-2167, or object-based design.

...for Software Developers

The transition to the use of structural modeling and the associated technologies represented a significant change for most software developers. The ASC/YW engineering office shared the risk of flight simulators using significant new technology. For example, they supplied the early projects with consultants in structural modeling, Ada83, and object-based design.

Most projects provided software developers with short courses in Ada83, structural modeling, and any new development tools. On-the-job training was the primary educational vehicle. Many later projects used architects and software developers from previous projects, thereby reducing their transition time. One project developed in-house training on effective styles for writing maintainable Ada83 programs [Glaize 91].

Software developers from the early structural modeling simulators reported a minimum of six months, and typically more, to feel comfortable with structural modeling. Once the architecture and design were established and a core of software developers understood how to apply structural modeling, all projects reported on the extreme ease of bringing new team members onto a project. Most reported only a few weeks before new developers were productive. They cited the standard definition and use of subsystem controllers, communication mechanisms, and coordination mechanisms as key contributors.

...for Managers

Most managers within the air vehicle software development group have made an effort to understand the technologies and its implications on their job. Most managers at the program level, however, have remained isolated from the technology. All projects reported that managers at the program level received little or no training in either the technologies themselves—structural modeling, Ada83, or object-based design—or the potential impact of the technologies on their jobs.¹⁵ Most of those interviewed reported the importance of all levels

of management throughout the project (system engineering, hardware engineering, and program level) having a greater understanding of structural modeling, particularly the importance

- and benefits derived from an effective architecture
- throughout the life cycle (including maintenance) of having a sound software architecture early in a project
- of prototyping to ensure the viability of new concepts or mechanisms

...for Bid, Proposal, and Acquisition Staff

All of the projects followed traditional contract vehicles. Cost, size, and schedule were based on the architecture of past simulators. One organization reported that they are now modifying their bid proposal process based on their use of structural modeling.

5.7 Summary

The flight simulator projects had many new technologies to deal with at the same time. These were

- Ada83
- DoD-STD-2167
- structural modeling
- new development environment tools

In the mid- to late 80s, all these technologies were immature. Actual use on large, industrial-strength applications was limited. Published experiences were limited, and there were few detailed user guidebooks or experienced consultants. The people and organizations using these technologies during this early stage had to forge new territory.

Additional time is required for such problem and solution discovery. All of those interviewed noted the lack of sufficient training and experimentation as a problem when structural modeling was first introduced. Management support and involvement can mitigate many of these risks associated with new technology. However, structural modeling and Ada83 were introduced as a technical solution within the software development group with limited involvement of program-level project management. This has slowed the time to mature structural modeling and its spread through the flight simulator community.

The more technology is integrated into all parts of a project's organizations (program level, software, hardware, systems), the greater the effectiveness and dissemination of the technology. Limited integration of structural modeling within the organization and project management was reported by those interviewed. Most raised this situation as a source of the many

¹⁵. The Air Force and the Software Engineering Institute are currently developing a course introducing structural modeling to acquisition managers.

issues they faced in trying to apply the technology within their respective organizations. Major examples of the low integration routinely cited included the following:

- Milestones and resource allocations for preliminary design and critical design review were not adjusted to allow for increased analysis and design activities that a prescriptive architecture approach, such as structural modeling, requires.
- Organizational structure did not align with the architecture.
- A specific organizational element was not dedicated to own and control the architecture on a continuous basis.

Overall, there was less attention to the organizational and people implications of the structural modeling technology than on the technical aspects.

6 Analysis and Conclusions

1 Introduction

Structural modeling has provided an effective solution to the problem of software complexity as experienced by the aircrew trainer community. This has consistently been verified by the interviews we conducted and by the related contractor publications. Structural modeling's effectiveness is further verified by its continued use on new trainers.

In this chapter, we will offer some additional observations on

- the limitations of this study
- organizational and technical factors contributing to the effectiveness of structural modeling
- structural modeling's relationship to object-oriented development
- a possible product line of simulation software based on structural modeling

6.2 Limitations of the Case Study

The major limitation of this study is the lack of publicly available quantitative data. Although several projects reported the collection of quantitative data, those data were either unprocessed or not publicly available. The evidence presented in this study is either the verbal or published experience (or opinion) of those interviewed, or is deduced from the software architecture.

6.3 Organizational Factors

The use of structural modeling on the B-2 project represented a drastic departure from the previous development method based on the waterfall model and the data-driven architecture. Two key factors strongly contributed to the success of the B-2 effort:

- The U. S. Air Force had a strong vested interest in the success of the aircrew trainer project; in particular, they wanted an early success with Ada on a large project. The DoD had mandated the use of Ada, and the Ada Simulator Validation Project (ASVP) was specifically tasked to demonstrate viability of Ada in this domain [Lee 89].
- Structural modeling was a technology developed and driven from the technical engineering arm of the ASC/YW office. Most successful technologies have had a champion who sees the value of a technology and, through various ways, leads a group to actively use the technology. With larger groups, the original champion is augmented with surrogate champions that work with particular subgroups to carry the message. We see the same trend with structural modeling. Typically, a surrogate champion has emerged within each flight simulator project and has worked to "sell" the technology within their group or company.

Together these factors represented strong customer support for the introduction of then new technology.

6.4 Architecture-Based Development

During the early stages of the B-2 trainer development, structural modeling referred to the architecture; structural modeling was a process for extracting the mechanisms for coordination and communication from the user-required functionality, and exploiting the commonalities discovered within those mechanisms. Later in the B-2 development, as the implications of the architecture generated by structural modeling became clearer, a more general view of structural modeling emerged, that of a development process supported by the architecture. That later view of structural modeling is as follows:

Structural modeling is the local cover name for the general engineering principles and technologies used: a prescriptive architecture, incremental development, and prototyping. The prescriptive architecture involves an object-based strategy, a small set of structural types, an explicit coordination model, and enforcement policies to ensure conformance to the architectural principles and guidelines. Incremental development is based on the use of prototypes to evolve the implementation in manageable deltas. Prototyping is used to discover the real system requirements, and then to validate those requirements and the resulting implementation.¹⁶

Structural modeling views the software architecture as the backbone of the entire software life cycle, not just of the software development effort. The architecture not only addresses the initial configuration of the software, but also provides a plan for dealing with the anticipated directions of change in the software. Further, it is the primary tool for communication among the system stakeholders.

Architecture-based development involves domain analysis, iterative development, effective representation and communication, and enforced system conformance to the software architecture. The relationship of structural modeling to each of these architecture-based development activities is described below:

- Domain Analysis: Although formal domain analysis was not practiced when structural modeling was developed, structural modeling separates the domain commonalities from variabilities in the architecture (executive vs. state-calculating components). It does not, however, separate domain commonalities from variabilities across the state-calculating components.
- Iterative architecture development and evaluation of the architecture: The architecture was developed and its performance evaluated using the integration harness.

¹⁶. This description was written by Joe Batman in a private communication.

- Effective representation and communication of architecture to stakeholders: The software components correspond to the parts of the aircraft, and the physical connections in the plane are explicitly represented and exposed in the software. Hence the software architecture provides a common language for the designers, implementers, and maintainers of the software and the designers and maintainers of the aircraft.
- Enforced system conformance to the selected architecture during system development and maintenance: Conformance to the software architecture generated by the structural model is enforced by (1) design and code reviews and (2) the use of software templates for the subsystem controllers and state-calculating components.

6.5 Structural Modeling and Object-Oriented Development

6.5.1 Overview of Object-Oriented Development

Object-oriented development is an incremental, iterative process involving [Booch 91]

- analysis of system requirements based on classes and objects derived from the problem domain
- design based on an object-oriented decomposition
- implementation based on cooperating objects and a hierarchy of classes related by inheritance

Object-oriented development is based on the principles of [Booch 91]

- abstraction: “the process of focusing on the essential characteristics of an object.”
- encapsulation: “the process of hiding all of the details of an object that do not contribute to its essential characteristics.”
- modularity: “the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.”
- hierarchy: “the ranking or ordering of abstractions”
- typing: “the enforcement of the class of an object...”

Booch lists software component reuse, changeability, and reduction of developmental risk as the major benefits of object-oriented development, while performance and start-up costs are listed as the major associated risks.

6.5.2 Similarities

There are strong similarities between structural modeling and object-oriented design techniques. In fact, the structural modeling effort started with a study to determine the benefits of using object-based concepts for aircrew trainer software [Lee 89]. Similarities include:

- an incremental, iterative development process
- the use of abstraction, encapsulation, modularity, and typing¹⁷
- the use of hierarchy
- the correspondence between classes and structural types
- the lack of hard evidence of the effectiveness of the approach¹⁸

6.5.3 Differences

The differences between structural modeling and current object-oriented development techniques are largely historical. The B-2 project was partly a proof of concept for the use of Ada83 for large-scale simulation software projects. Ada83 supports object-based, but not object-oriented development. The difference between object-based and object oriented programming lies in the implementation programming language's direct support of inheritance.

Although the air vehicle implementation is clearly object-based, an implementation using an object-oriented language such as C++ or Ada95 could define

- abstract classes corresponding to the air vehicle structural types
- corresponding concrete classes to capture further commonalities across groups of similar components (e.g., pumps)

The change to an object-oriented language would

- obviate the need for the software templates and the interface management (as reported by the B-2 project), as these would be language-supported
- promote further reuse by the exploitation of functional commonalities across similar groups of state-calculating components

17. Typing in the air vehicle is not based on classes and inheritance, but rather on the structural types.

18. From a lecture by Adele Goldberg at the Tenth European Conference on Object-Oriented Programming, in Linz, Austria in July, 1996.

6.6 Product Line of Air Vehicle Simulation Software

A product line is a group of related systems that address a market niche or mission, and is built from a common set of assets, such as software components, personnel, project planning expertise, performance analyses, processes, methods, tools, and exemplar systems [Withey 96]. The potential benefits of a product-line approach include reuse of assets; reduction of risk, cost, and schedule predictability; and reduction in cost and time to market [Brownsword 96].

The software architecture forms the technical foundation for a product line [Brownsword 96]. Architecture-based development is “a product-line strategy where the earliest focus is on defining reusable design structures, together with technical practices governing their use as the basis for individual products.”¹⁹ In fact, one contractor interviewed reported the successful use of structural modeling for designing for reuse. A product line of air vehicle simulators based on structural modeling seems a natural next step. The U. S. Air Force is investigating the feasibility of such a product line.²⁰

The nontechnical issues (including business, organizational, personnel, management practice, cultural, and process issues) are as critical to the success of the product line as the technical issues [Brownsword 96]. These issues are further complicated by the current Air Force acquisition process. ASC/YW, as the acquisition agent, has been involved with each structural modeling-based simulator project; however, each of those projects to date has also involved a separate program office, a development contractor, and frequently a different maintenance contractor.

A product line approach to acquisition must address such questions as the following:

- How can a product line be identified and managed at the Air Force level?
- How can contractors for a product line be identified and managed?
- Who owns the product-line assets?
- How can those assets be effectively managed across multiple contractors and program offices?
- Can the present acquisition process effectively deal with product lines?

19. From a private correspondence from Larry Howard.

20. From the Software Engineering Special Report *Product Line Identification for ESC-Hanscom* (CMU/SEI-95-SR-024) by Sholom Cohen, Seymour Friedman, Lorraine Martin, Nancy Solderitsch, and Robert Webster.

Certainly a different approach to software acquisition is required if the Air Force is to realize the benefits of a product-line approach. The Air Force Electronics Systems Center (ESC) report, *Concept of Operations for the ESC Product Line Approach*, [Cohen 96] contains

- a proposed organization implementing a product-line approach to acquisition
- descriptions of the roles and responsibilities of those organizations
- key issues related to the transition to a product line approach to acquisition

Appendix A The Executive

A.1 Introduction

The air vehicle simulator is typically implemented as multiple processes on multiple, distributed processors. The executive binds the group of subsystems within a given process, handling the scheduling of those subsystems. The executive(s) are responsible for coordination of the multiple processes within the air vehicle implementation, external (to a system) communication of state information, and the overall synchronization of a system with the rest of the simulator.²¹

The executive is partitioned by function, into software that supports periodic control, aperiodic control, external communication, and overall control and coordination. Periodic control is handled by the periodic sequencer, which is responsible for invoking the appropriate state calculations in the proper order. Aperiodic control is handled by the event handler, which is responsible for determining and then executing the appropriate code in response to an event (e.g., a request from the IOS). External periodic communication is handled by the surrogate, which is responsible for hiding the details of external state information communications. The overall system is controlled by the time line synchronizer, which is responsible for synchronizing the system with the rest of the simulator, and scheduling and invoking the periodic and aperiodic control software (see Figure 10). These four partitions of the executive are described in the following sections.

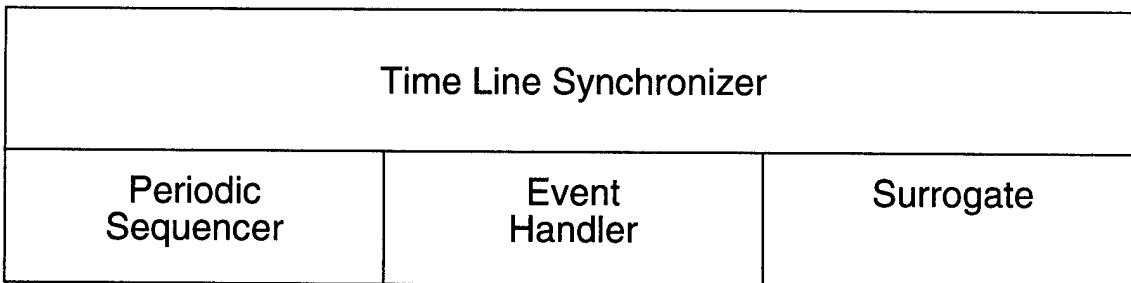


Figure 10: Executive Partitioning

A.2 Periodic Sequencer

The periodic sequencer is responsible for scheduling the periodic tasks within the air vehicle (Figure 11). It invokes the subsystem controllers according to a previously computed, fixed scheduling table, thereby ordering the execution of its associated subsystems. The periodic sequencer is also responsible for coordination of the accessing and updating of the export areas of its associated subsystem controllers

²¹. In this paper, we refer to an executive and its associated subsystems as a system. The air vehicle is typically implemented as a set of systems.

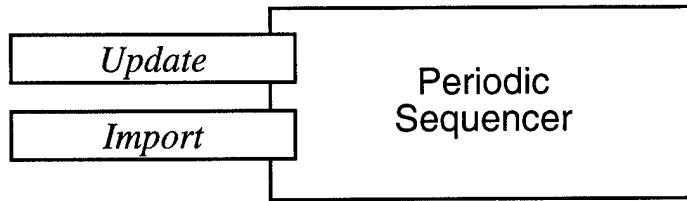


Figure 11: Periodic Sequencer

The operations of the periodic sequencer are *import* and *update*. *Import* invokes the *import* operations of the appropriate subsystem controllers, while *update* invokes the *update* operations of the appropriate subsystem controllers.

A.3 Event Handler

The event handler is responsible for event processing within the air vehicle (Figure 12). It determines which subsystem controllers should receive an event, and invokes the appropriate subsystem controller to handle that event. Events, primarily generated by the IOS, are externally placed in an event queue, which is later processed by the event handler. Aperiodic events are relatively rare; hence the event handler eliminates the need for polling in the subsystems.

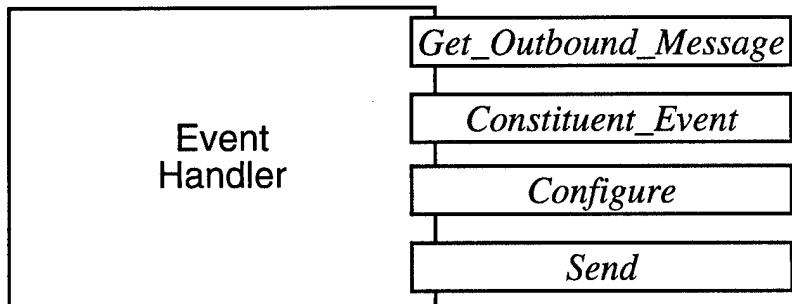


Figure 12: Event Handler

The operations of the event handler are *get_outbound_message*, *constituent_event*, *configure*, and *send*. *Get_outbound_message* retrieves, from the event queue, a message bound for outside of the system; *constituent_event* routes an event to the appropriate subsystem controller; *configure* invokes the appropriate subsystem controllers' *configure* operation; and *send*, called by subsystem controllers, places an event in the event queue.

A.4 Surrogate

The surrogate is responsible for the periodic exchange of state information between systems (Figure 13). Surrogates function in pairs. For each external system that a local system communicates with (i.e., exchanges state information with), there is a surrogate on the local sys-

tem and an associated surrogate on the external system. Surrogates hide the details of the connections between systems, such as data representation differences between processors.

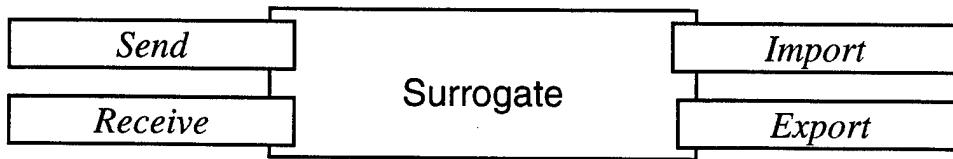


Figure 13: Surrogate

The operations of the surrogate are *send*, *receive*, *import*, and *export*. *Send* transfers state information from the surrogate to its paired surrogate in the receiving system; *receive* accepts state information from the paired surrogate in the sending system; *export* makes the received external state information available to the system; and *import* instructs the surrogate to prepare the state information to be sent.²²

A.5 Timeline Synchronizer

The timeline synchronizer is responsible for scheduling and invoking the periodic sequencer, the event handler, and the surrogate. It schedules based on the fixed period of time that its system is to execute, as shown in Figure 14.

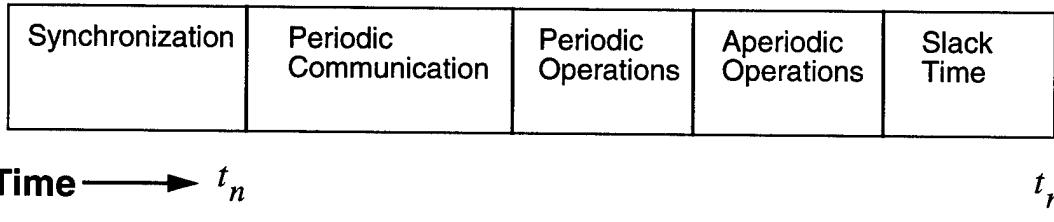


Figure 14: System Time Period

The timeline synchronizer is also responsible for maintaining the system's view of the simulator's overall state. The timeline synchronizer has no visible operations.

The timeline synchronizer coordinates the multiple systems within the air vehicle. The system that executes at the highest rate assumes the role of the master timeline synchronizer, and synchronizes the execution of the multiple systems using some inter-process synchronization mechanism, such as semaphores [Abowd 94].

²² The import and export operations are from the point of view of the surrogate. Hence, the surrogate exports its received state information to the rest of the surrogate's system.

Appendix B Control Flow

The flow of control in the air vehicle is for the most part top down.²³ For example, the timeline synchronizer calls the periodic sequencer as needed, but the periodic sequencer does not call the timeline sequencer. Figure 15 illustrates the flow of control during a periodic update of the air vehicle system, where an arrow from box A to box B indicates that the code associated with box A calls the code associated with box B. The box associated with the periodic sequencer is shaded to emphasize the periodic sequencer's role in periodic processing. Figure 16 similarly illustrates the flow of control in response to an event.

Figures 15 and 16 also illustrate a portion of the T39A air vehicle architecture [ASCYW 94].²⁴ The boxes labeled timeline synchronizer, periodic sequencer, and event handler all represent specific instances of their respective structural types and are tailored for the specific processor on which the represented system would execute. The hydraulic power subsystem controller is an instance of the subsystem controller structural type that coordinates the execution of the state-calculating components for the hydraulic pump, reservoir, and control relay. Other components of that subsystem, including a shut-off valve and a fluid filter are not shown. The hydraulic panel subsystem similarly simulates the operation of the hydraulic system's displays, indicators, and controls.

- 23. The exception is the event handler's *send* operation, whereby a subsystem controller places an event on the event queue.
- 24. The T39A is a jet aircraft used for training by the U. S. Air Force.

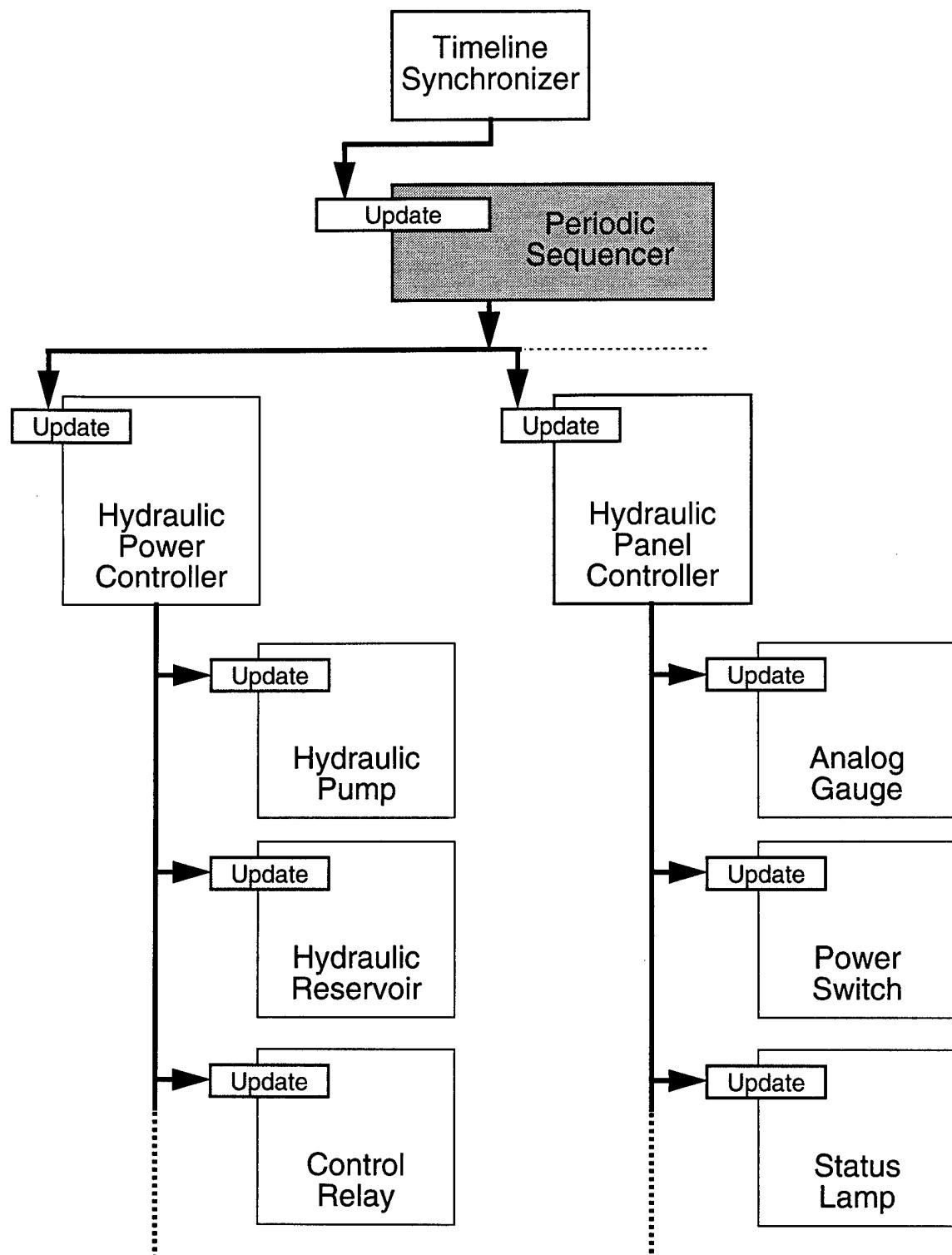


Figure 15: Flow of Control: Periodic

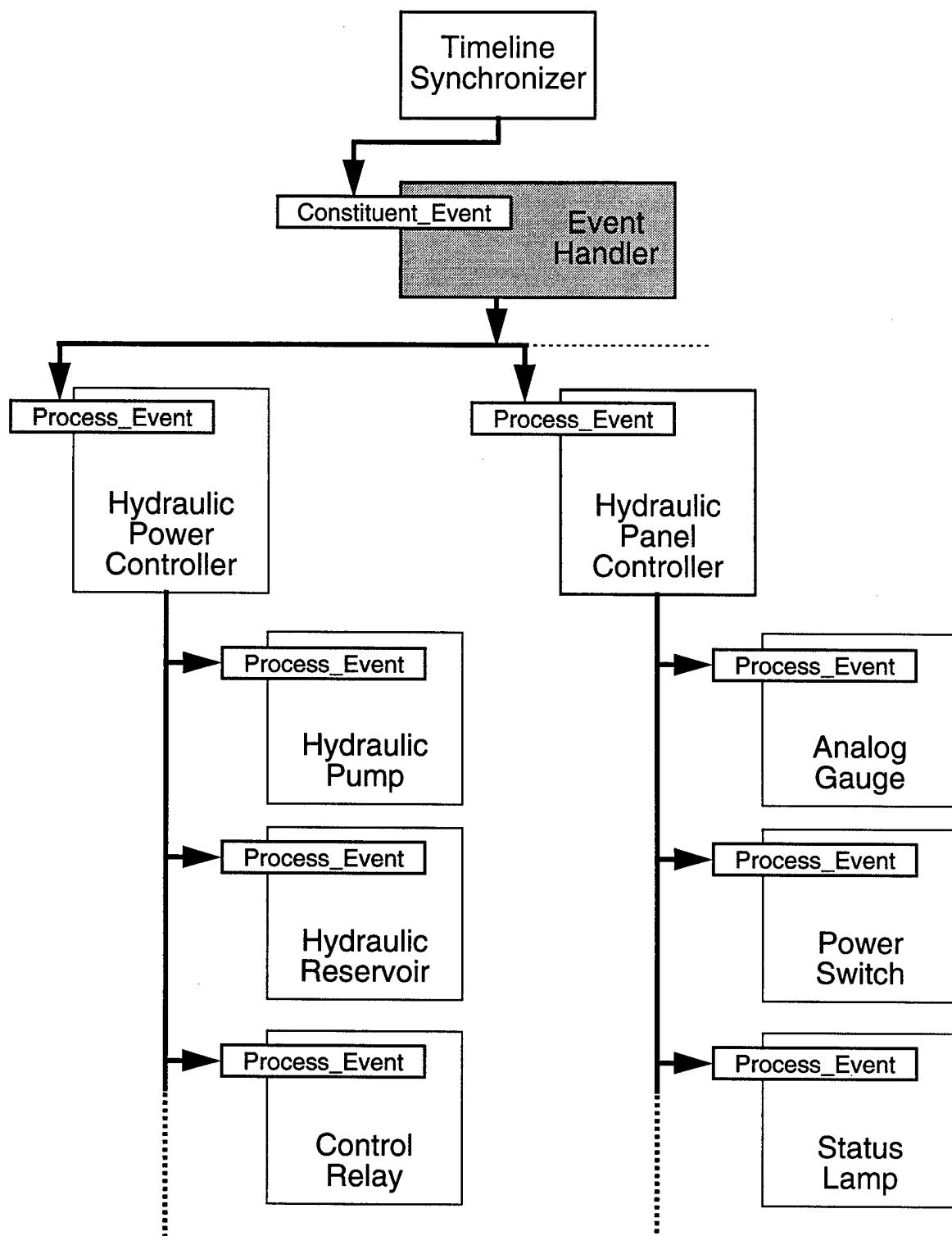


Figure 16: Flow of Control: Aperiodic

References

Cohen 96 Cohen, Sholom; Friedman, Seymour; Martin, Lorraine; Royer, Tom; Solderitsch, Nancy; & Webster, Robert. *Concept of Operations for ESC Product Line Approach* (CMU/SEI-96-TR-018, ADA310914). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

Garlan 95 Garlan, David & Perry, Dewayne. "Introduction to the Special Issue on Software Architecture (Guest Editorial)," *IEEE Transactions on Software Engineering* 21, 4(April 1995): 269-274.

Glaize 90 Glaize, John. "Organizational Barriers to Object-Oriented Development," 362-367. Interservice/Industry Training Systems Conference. Orlando, FL, November 1990.

Glaize 91 Glaize, John. "Experiences in Writing Readable and Understandable Ada," 1-7. Interservice/Industry Training Systems Conference. Orlando, FL, December 1991.

Kruchten 95 Kruchten, Philippe B. "The 4+1 View Model Of Architecture." *IEEE Software* 21, 11 (November 1995): 42-50.

Lee 89 Lee, Kenneth J. & Rissman, Michael S. *An Object-Oriented Solution Example: A Flight Simulator Electrical System* (CMU/SEI-89-TR-5, ADA219190). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1989.

Rolfe 86 Rolfe, J.M. & Staples, K.J. *Flight Simulation*. New York, NY: Cambridge University Press, 1986.

Stikeleather 96 Stikeleather, James. "The Importance of Architecture," *Object Magazine*, 6, 2 (April 1996):20-24.

Withey 96 Withey, James. *Investment Analysis of Software Assets for Product Lines* (CMU/SEI-96-TR-10, ADA304091). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS None	
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release Distribution Unlimited	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-96-TR-035		5. MONITORING ORGANIZATION REPORT NUMBER(S) ESC-TR-96-035	
6a. NAME OF PERFORMING ORGANIZATION Software Engineering Institute		6b. OFFICE SYMBOL (if applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI Joint Program Office
6c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213		7b. ADDRESS (city, state, and zip code) HQ ESC/AXS 5 Eglin Street Hanscom AFB, MA 01731-2116	
8a. NAME OFFUNDING/SPONSORING ORGANIZATION SEI Joint Program Office		8b. OFFICE SYMBOL (if applicable) ESC/AXS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-95-C-0003
8c. ADDRESS (city, state, and zip code) Carnegie Mellon University Pittsburgh PA 15213		10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO 63756E PROJECT NO N/A TASK NO N/A WORK UNIT NO N/A	
11. TITLE (Include Security Classification) A Case Study in Structural Modeling			
12. PERSONAL AUTHOR(S) Gary Chastek, Lisa Brownsword			
13a. TYPE OF REPORT Final	13b. TIME COVERED FROM TO	14. DATE OF REPORT (year, month, day) December 1996	15. PAGE COUNT 56
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (continue on reverse if necessary and identify by block number) aircrew trainer simulation software, data-driven architecture, software architecture, structural modeling	
FIELD	GROUP	SUB. GR.	
19. ABSTRACT (continue on reverse if necessary and identify by block number) This report is one in a series of Software Engineering Institute (SEI) case studies in software architecture. It describes structural modeling, a technique for creating software architectures based on a small set of design elements called structural types. Structural modeling resulted from the efforts of the Air Force Aeronautical Systems Command (ASC/YW) and has been used by Air Force contractors since the late 1980s to design large-scale, high-fidelity aircrew trainer simulation software. This report examines the changes, resulting from the use of structural modeling, to the trainer's software architecture and to the development methods used.			
(please turn over)			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS <input checked="" type="checkbox"/>		21. ABSTRACT SECURITY CLASSIFICATION Unclassified, Unlimited Distribution	
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas R. Miller, Lt Col, USAF		22b. TELEPHONE NUMBER (include area code) (412) 268-7631	22c. OFFICE SYMBOL ESC/AXS (SEI)

ABSTRACT — continued from page one, block 19